UNITED STATES PATENT APPLICATION FOR

SYSTEMS AND METHODS FOR A COMMON RUNTIME CONTAINER FRAMEWORK

Inventor:

Kyle Marvin

CERTIFICATE OF MAILING BY "EXPRESS MAIL"
UNDER 37 C.F.R. § 1.10

"Express Mail" mailing label number: **EV 386447431 US**
Date of Mailing: 2/11/04, 2004

I hereby certify that this correspondence is being deposited with the United States Postal Service, utilizing the "Express Mail Post Office to Addressee" service addressed to **Mail Stop PATENT APPLICATION, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450**, and mailed on the above Date of Mailing with the above "Express Mail" mailing label number.

Tina M. Galdos
Signature Date: 2/11/04, 2004

SYSTEMS AND METHODS FOR A COMMON RUNTIME CONTAINER FRAMEWORK

Inventor: Kyle Marvin

COPYRIGHT NOTICE

CLAIM OF PRIORITY

[0002] This application claims priority from the following application, which is hereby incorporated by reference in its entirety:

[0003] U.S. Provisional Application No. 60/451,012, entitled SYSTEMS AND METHODS FOR A COMMON RUNTIME CONTAINER FRAMEWORK, by Kyle Marvin, filed on February 28, 2003 (Attorney Docket No. BEAS-01399US0).

FIELD OF THE INVENTION

[0004] The present invention relates to software runtime containers and software frameworks.

BACKGROUND

[0005] The use of software containers provides several advantages when developing and deploying software applications. Containers provide software developers and users with a high level of abstraction. In other words, containers provide software functionality that can be neatly packaged and delivered while hiding significant implementation complexity. As an example, container packaging is often used to develop and deploy modular units of "pluggable" software functionality known as controls. In spite of these advantages, the design, construction, and deployment of containers at runtime have been a complex process, requiring software developers to have a large skill set.

BRIEF DESCRIPTION OF THE DRAWINGS

[0006] **Figure 1** shows an overview of one exemplary framework for a common runtime container that can be used in accordance with one embodiment of the present invention.

[0007] **Figure 2A** is one possible example showing the inheritance structure of a set of runtime containers that can be used in accordance with one embodiment of the present invention.

[0008] **Figure 2B** is one possible example showing an inheritance structure of a set of metadata object classes that can be used in accordance with one embodiment of the present invention.

[0009] **Figure 3** illustrates the duality between the metadata objects and the containers in accordance with one embodiment of the present invention.

[0010] **Figure 4** illustrates an exemplary architecture for a common runtime container that can be used with some embodiments.

DETAILED DESCRIPTION

[0011] The invention is illustrated by way of example and not by way of limitation in the figures of the accompanying drawings in which like references indicate similar elements. It should be noted that references to "an" or "one" embodiment in this disclosure are not necessarily to the same embodiment, and such references mean at least one.

[0012] While details of certain embodiments are discussed in this section, it should be clear that other suitable embodiments exist and can be used to achieve similar capabilities. Further, some of these embodiments may include additional functionality not discussed herein, and/or may not contain all of the functionality described herein.

[0013] Systems and methods in accordance with embodiments of the present invention provide a complete framework useful in the design, construction, and deployment of software runtime containers, useful to both software developers and end users. Such a framework can allow runtime containers to inherit the functionality and runtime environment attributes of other containers. Having a single runtime container framework, which is extensible to support multiple component types, can provide benefits such as the following:

- Allowing common runtime containers to inherit functionality from other common runtime containers.
- Creating new component types and adding incremental features more efficiently via a common architecture and leveraging existing code leverage. Divergent implementations of common features are costly to implement in parallel and difficult to synchronize with enhancements over time.

- Ensuring behavioral compatibility across component types for common programming model features. This creates a uniform development environment reducing the knowledge and skills required of each developer using the environment. A common programming model involves more than syntax. For example, the common runtime container can support and simplify the development and deployment of controls.
- Providing a runtime environment with uniform and maintainable deployment, management and services capability.
- Providing a common set of container infrastructure and services (application generation, deployment, logging, debugging, test harness, etc.). A common container expands the leverage of capabilities built around each container.

[0009] Services inherited from the use of a common runtime container framework can include, but is not limited to:

- Uniform state, context and event management services, possibly using a metadata scheme, in the runtime environment.
- Synchronous/Asynchronous Invocation and uniform event management.
- A container environment to host controls.
- Uniform capabilities for application generation and deployment in both the development and runtime environments.
- Context services and event management.
- Common Configuration model for multiple input protocols and messaging architectures (i.e. SOAP, HTTP, XML schemas).
- A uniform set of security services.

[0010] An overview of one exemplary framework for a common runtime container is shown in **Figure 1**. Within this exemplary framework **100**, the external routing and event handling services or components **101** can be used for communications of requests and responses with external client entities. These services or components can communicate with an invocation component **102** within the runtime container. In some embodiments, these services or components will use a uniform or standardized protocol to communicate with the container. The use of a uniform interface allows for the support of a large number of external interfaces without the need for explicit support for more than one type of interface.

[0011] Within the exemplary framework for a common runtime container, invocation component can receive requests and dispatch them to the correct interfaces of service components **110** inside

the container **104**. This dispatch process can be dependent on the nature of the request, the state information, and context information or other information stored in metadata **106.** The metadata herein includes context, state, and/or other information about the data and objects being processed upon the requests. The invocation component can route or manage the returned responses as required, based on the nature of the response, the state information, and context information or other information stored in the metadata.

[0012] The container can invoke the services provided by the components within the container to process requests and produce responses. The container can provide state information and context information to these components at runtime. If external services **108** are used during the processing, those services can be engaged through one or more interfaces provided by a control component **107**. For embodiments using the Java programming language, the components can be created in the form of Java beans.

[0013] Within such a framework, the invocation component and the container can receive both state information and context information. The invocation component or the container can query the state manager **105** to retrieve state from nonvolatile storage **109**. Likewise, the invocation component or the container can query the context manager **103** to receive context or other information from the metadata.

[0014] Services provided by the components in the common runtime container can perform either preprocessing or post-processing of requests and responses sent to or returned from the hosted components. For example, these processing services can include the processing of message protocols (e.g. SOAP), and request management (e.g. the tracking of session IDs or callback management). The preprocessing and post-processing of messages can be decoupled in that the preprocessing may not depend on post-processing and vice versa.

[0015] In some embodiments, common runtime containers are extensible via a hierarchical architecture, i.e., they can be created from a container class. The services of the runtime containers can be exposed through the methods of the container class. Contents of the containers can be accessed through the containers' interfaces. These common runtime containers can be sub-classed through an inheritance mechanism. Using inheritance mechanisms, developers can extend the functionality of common runtime containers. For example, methods and attributes can be inherited from one container to another and extended as required. These common runtime containers can be nested so that containers can inherit functionality and properties from other containers through several levels. Software components, typically supplied by the application developer or user, can be contained within or wrapped by the one or more runtime containers.

[0016] One possible example showing the inheritance structure of a set of runtime containers is shown in **Figure 2A**. In this example, a Web services container **201** inherits from the base container **200**, and the workflow container **202** inherits from the Web services container. Other containers can follow arbitrary complex inheritance structures. In no case should this particular example imply a limitation on the scope, functionality or spirit of the invention.

[0017] At the same time, metadata, including context information used by the containers and components, is contained within one or more metadata objects. The metadata objects can also be subclassed through a hierarchical architecture. These objects can be nested so that metadata objects can inherit properties from other metadata objects. The metadata object is of a metadata class (the base class can be referred to Metadata **210**) and can have methods (metadata methods) and interfaces (metadata interfaces), that are generally used to get or set metadata values. In some embodiments, the root metadata object is a singleton.

[0018] One possible example showing an inheritance structure of a set of metadata object classes is shown in **Figure 2B**. In this example, a Web services metadata object **211** inherits from the base metadata object **210**, and the workflow metadata object **212** inherits from the Web services metadata object. Other metadata object objects can follow arbitrary complex inheritance structures.

[0019] In some embodiments, metadata objects and containers are organized in a duality, wherein containers at one level in the hierarchical architecture can access metadata contained in the metadata objects at that same level in the hierarchical architecture. In other words, there can be a mapping between the metadata objects and the containers. This duality is illustrated in **Figure 3**. The arrows connect the containers at each level to the metadata objects at the same level. In this example, a Web services container **305** inherits from the base container **304**, and a Web services metadata object **302** inherits from the base metadata object **301**. The workflow container **306** inherits from the Web services container, and the workflow metadata object **303** inherits from the Web services metadata object **302**. This type of structure can be useful in cases where a workflow application is exposed as Web services, for example. It can be seen that the structure or hierarchy of the objects in this example follow that shown in **Figures 2A** and **2B**.

[0020] In some embodiments, a well-defined Application Programming Interface (API) can be used to create more levels in the runtime container and metadata object hierarchies. Extensions to this API can allow developers to create new types of containers or customize existing containers with incremental features. In some embodiments, the API will be in the form of one or more public interfaces to the container class. As an example, a "factory pattern" can be used to create

other levels in the hierarchy of runtime containers or metadata objects. The created component types can be pluggable in some cases.

[0021] As has already been discussed herein, a common runtime container can provide invocation services using components contained therein. Such services can traverse several layers in the hierarchy of runtime containers. The invocation component can encompass both pre-invoke (for processing requests) and post invoke (for processing responses). As an example, some embodiments will use a process such as the following to pass control from one level to the next. Referring to the exemplary structure shown in **Figure 3**, the following sequence of steps may occur:

1. A request is received in the runtime environment, which manages any message routing.
2. If the request constitutes the start of a new session, the runtime environment creates a new instance of the runtime containers and the components, possibly using descriptor information contained in the metadata objects.
3. The request is passed to the workflow container, which inherits the thread of control.
4. If this is a new session, requiring security techniques, the runtime containers can initiate and manage the security at any level in the hierarchy. Supported security techniques can include secure sessions, authentication, and role management.
5. Based on a unique identifier, specific to the session at that level in the container hierarchy, the container queries the workflow metadata object (the metadata object for that level of the hierarchy), which returns context or state (if any) information. In this example, the metadata can correspond to the state and context of the workflow session. It should be noted that in some embodiments, the request message might contain a unique, but different identifier for each level of the runtime container hierarchy. In other embodiments, a single identifier may be used. In yet other embodiments, one level in the hierarchy may create and use a unique identifier for the next level in the hierarchy. In some embodiments, the unique identifiers can correspond to the identifiers used by the one or more factories for different container objects.
6. Based on the session's unique identifier and the metadata, any required preprocessing is performed by the container. Examples of preprocessing can include, parsing and processing of message headers (e.g. HTTP and SOAP protocol headers), verification of security information or security tokens, callback management, message routing to one or more destinations, transformation of variables in the request message payload, etc. In

some embodiments, the work performed during preprocessing can be defined by a prerequisites list, possibly contained in the metadata.

7. The request is passed to one or more of the interfaces of the Web services container, which inherits the thread of control. This dispatch operation is based on state (if any) and context information derived from the metadata and the nature of the request.

8. The Web services container performs processing similar to steps 5 and 6. In this example, the metadata can correspond to the state and context of the Web services session.

9. The request is passed to one or more of the interfaces of the base container, which inherits the thread of control. This dispatch operation is based on state and context information derived from the metadata and the nature of the request.

10. The base container performs processing similar to steps 5 and 6.

11. Based on the nature of the request, and the context and state (if any) determined from the metadata and the unique session identifier (for the session at the level of the base container), the base container invokes the component interfaces required to process the request and passes the request to the components.

12. The components perform the processing for the request, based on the unique session identifier, the contents of the request, and the context and state (if any) derived from the metadata, and return the results to the base container possibly through a callback operation.

13. If external services are required to process the request, possibly using controls, the common runtime container manages any callback operations, sessions, security, message routing, or other services required to perform this external processing. In some embodiments, the base container will perform the dispatch operation to the external services, managing the response path or callback based on the information in the request and the metadata.

14. The base container receives the response from the components.

15. The based container saves any state or context as required.

16. The base container performs any required post processing on the response based on the unique session identifier and metadata, and passes the response to the Web services container possibly through a callback operation. In some embodiments, the work performed during post-processing can be defined by a prerequisites list, possibly contained in the metadata.

17. The Web services container inherits the thread of control and performs any required post-processing on the response based on the unique session identifier and metadata, and passes the response to the workflow container. In this example, the metadata can correspond to the state and context of the Web services session. In some embodiments, the work performed during post-processing can be defined by a prerequisites list, possibly contained in the metadata.

18. The workflow container inherits the thread of control and performs any required post-processing on the response based on the unique session identifier and metadata, and passes the response to the runtime environment, possibly through a callback operation. In this example, the metadata can correspond to the state and context of the workflow session. In some embodiments, the work performed during post-processing can be defined by a prerequisites list, possibly contained in the metadata.

19. The runtime environment routes the response, which can be in the form of a callback operation.

20. If the session is being terminated, the runtime environment destroys the runtime container objects and their contents.

[0024] An exemplary architecture that can be used with some embodiments for a common runtime container is shown in **Figure 4**. In embodiments using the Java programming language, this architecture can be built in the J2EE runtime environment. Other architectures can be applied in alternative embodiments.

[0025] Starting at the left hand side of **Figure 4**, one or more servlets **402-404** are used to manage communications between the common runtime container **400** and external entities such as clients. A servlet is a computer program (such as a Java Bean) that can run on a computer and provide certain kind of services. The servlet **402** is typically associated with a listener **401** that monitors incoming communications on the external interface of the servlet. Clients or other external entities can send requests and receive responses, using one or more protocols with servlets capable of processing those protocols. As an example only, communications with a servlet can include TCP/IP, HTTP, SOAP, and perhaps application specific (e.g. an XML schema) protocols. Specific servlets can be created to support specific sets of protocols. In some embodiments, the servlets communicate with a dispatcher component **405** using a common or uniform protocol. Thus, the servlets can translate both requests and responses between wide varieties of "wire" protocols and common a communication protocol used by the dispatcher. In some embodiments, the servlets can use a proxy for this communication. Some exemplary

protocols include JMS and CORBA messages, and clearly many more are possible. Not all protocols need to be transformed by the servlets. For example, an application specific protocol in the payload of a message may need to be passed unaltered to the application-specific components in the common runtime container. In some embodiments, the servlets can be stateless and synchronous. In some embodiments, the servlets can be modeled as top-level controls.

[0026] Still referring to **Figure 4**, a first dispatcher **405** receives and processes requests sent from one or more servlets. In some cases, the dispatcher receives direct requests from other components within the same process space. The dispatcher will determine which components to invoke based on the contents of the request and information on context retrieved from metadata (Meta) **406**. Requests requiring asynchronous processing are dispatched to the queue **407**. Requests requiring synchronous processing can be routed directly to a stateful component (for stateful processing) **410** or a stateless component (for stateless processing) **416**. A second dispatcher **408** receives the asynchronous requests dispatched to the queue **407**. The dispatcher will determine which components to invoke based on the contents of the request and information on context retrieved from metadata **409**. In some embodiments, the queue can follow a FIFO scheme, while some other embodiments can allow requests to be ordered by priority. This second dispatcher can dispatch requests requiring synchronous processing be routed directly to a stateful component (for stateful processing) or a stateless component (for stateless processing). In some embodiments using the Java programming language, the dispatcher can be in the form of an Enterprise Java Bean (EJB).

[0027] Still referring to **Figure 4**, stateless processing can be performed by one or more stateless components. The stateless component **416** can derive context information (e.g. information related to the session) from the metadata **419**. A stateless component can contain an arbitrary amount of code for the processing logic **417**. In many cases this logic is application-specific. A stateless component can call other stateless components within the common runtime container as part of a processing chain. A stateless component can call one or more external services **420**. In some embodiments, synchronous or asynchronous controls **418** associated with the component are used for communications with these external services. Other architectures can be used in other embodiments. In some embodiments using the Java programming language, these components can be constructed as Java Beans.

[0028] Still referring to **Figure 4**, stateful processing can be performed by one or more stateless components. The stateful component **410** can derive context information (e.g. information related to the session) from metadata **412**. State information is retrieved from nonvolatile storage **415**

through a state management component **414**. In some embodiments, the state information can be stored in the form of one or more Binary Large Objects (BLOBs). In other embodiments, the state information can be stored in relational tables in a Relational Database Management System (RDBMS). In yet other embodiments, both BLOBs and relational tables can be used. A stateful component can contain an arbitrary amount of code for the processing logic **411**. In many cases this logic is application specific. A stateful component can call stateless components or other stateful components within the common runtime container as part of a processing chain. A stateful component can call one or more external services **420**. In some embodiments, synchronous or asynchronous controls **413** associated with the component are used for communications with these external services. Other architectures can be used in other embodiments. In some embodiments using the Java programming language, these components can be constructed as Java Beans.

[0029] In some embodiments, the following process can be used to generate and use the common runtime container and other associated software. User supplied source code can be compiled in the usual manner with a suitable complier. The automatic code generation and class assembly for the container, servlets, components and controls can be performed at load time. In some cases, metadata or deployment descriptors created by the developer or system administrator are used to guide the automatic code generation and assembly process. In some embodiments, the runtime environment may perform configuration verification at runtime to prevent problems with configuration mismatch to runtime environment. When a new session is initiated, an instance of the containers and associated components and controls are generated. Processing for this session is then carried out as has already been described for the duration of the session. When the session completes, the containers and their contents are destroyed and the process space and system resources are reclaimed.

[0030] In some embodiments of the present invention, the invocation component in a common runtime container framework can be event-driven. Event delivery (calls in from a primary client or callbacks from controls) to a component as well as event generation (callbacks to a primary client or calls into controls) from a component can be synchronous or asynchronous. This invocation component can include synchronous and asynchronous capabilities such as the following:

- Base mechanism for locating the dispatcher associated with a component. For example, in a Java environment, the JVM cache may be used. Alternatively, a two-

level cache backed by JNDI to support remote lookup/invocation as well as local dispatch can be used.

- Ability to construct requests defining dispatchable operations or callbacks that target a specific conversation or control instance. The request interface can be an abstract interface supporting variability in internal request wire protocols, flexible mapping from wire protocol to class method signatures, and facilities for streaming requests.
- Ability to dispatch requests synchronously using local method calls or asynchronously by enqueuing on an asynchronous work queue for the service.
- Well-defined asynchronous request retry, delay, and exception handling semantics that allow container and/or user code to fire on asynchronous request failure.
- Preservation of authentication context for both synchronous and asynchronous request dispatch. For example, for asynchronous invocation, the principal at time of enqueue can be restored when the request is dispatched asynchronously.
- Definition and implementation of a more general mechanism for component dispatcher identity and lookup. For example, in a Java language environment, a JNDI-based scheme based on the current ServiceHandle interface can be used.

In some embodiments, there can be a clear separation of the generic container set of class implementation interfaces from, for example, Web services or other specific services interfaces. For example, in some Java language environments, a generic set of container interfaces (e.g. Stream_Web_Services_Request) is separated from Web service container specific subclasses (Soap-Web_Services_Request, MimeXmlRequest). Some embodiments supply retry/failure exception handling semantics for failed requests, which can expose user code.

[0031] In some embodiments of the present invention, the runtime container in a common runtime container framework can define simplified component abstractions that are transparently mapped (perhaps through code generation and assembly) to a more complex set of runtime environment components to create and deploy applications. For example, in some Java enterprise application environments, the runtime environment components are associated with the J2EE platform which hosts the Java beans comprising the application. This runtime container to generate and deploy application services can include capabilities such as the following:

- Generating appropriate runtime environment modules containing components required to support dispatch, state management, and all embedded controls. In some J2EE environments, these modules will be in the form of Enterprise Java Beans (EJB).

- Merging the generated modules into a larger application for the runtime environment, which might consist of both component-specific modules (e.g. http listening for a Web services application) and user-developed modules (e.g. EJBs or Web services code).
- Supporting a development mode where the larger application for the runtime environment is developed and packaged (e.g. and exploded EAR format for J2EE applications) into internally generated modules that are hidden from the developer's view of the application.
- Supporting fast, iterative development and test at the level of a single generated module.
- Supporting the generation of a production deployable packaged application (e.g. an EAR file) using the sources and components found within the development mode environment (e.g. an exploded EAR).

[0032] In some embodiments of the present invention, the runtime container in a common runtime container framework can expose a common configuration model for how components and control attributes are specified declaratively and programmatically by the developer, as well as providing a common model for declarative configuration override at application deployment time. A common configuration model provided by the runtime container can include capabilities such as the following:

- Defining a base mechanism by which component (e.g. a service or control) configuration is loaded into memory based upon metadata contained within the class file and possibly derived from annotations.
- Providing an override model allowing a subset of the component configuration that is dynamically configurable using an external configuration mechanism such as a property file or descriptor.
- Providing an API set allowing a component or control type implementation to access its own configuration state at runtime. This could be either push (initialization time) or pull (run-time) based.
- Providing a possibly dynamic, override model for reconfiguration.
- Defining and implementing the interfaces for run-time access to configuration data.

[0033] In some embodiments of the present invention, the state manager in a common runtime container framework can provide the ability to develop stateful components where the physical

mechanisms used to locate, manage, and persist state are largely transparent to the developer. This state manager can include capabilities such as the following:

- Looking up state instance based upon key value(s).
- Mapping between internal persistence formats and the user view of state.
- Concurrency protection (i.e. serialization of requests and events).
- Multiple state-persistence formats offering varying degrees of longevity, durability, and transactional attributes of state (from in-memory to transacted DB).
- Aging/idle timeout of state instances.
- Visibility and versioning (i.e. ability to externally visualize and manage state).
- Extensible state mechanisms (i.e. the ability to store component internal state along with user state).
- Persistence using in-memory state for higher performance/short-lived conversations.
- Fine-grained control over internal persistence formats (for visibility and versioning).
- Base level support for secondary conversation keys (i.e. allowing a client or control to define its own key that maps back to the primary key space).
- Fine-grained control over state characteristics on a per-event basis (i.e. read-only methods, narrowing of read/write for partial state, etc).
- Mechanisms for how a specific component type can extend base user state to add in component-specific internal state.

[0034] In some embodiments of the present invention, the context manager in a common runtime container framework can expose component-level services and events to the programmer. This context manager can include capabilities such as the following:

- Allowing a component type implementation for defining a context interface, which may expose container-specific services to the developer.
- Providing hooks allowing the component type implementation to perform container-specific processing at various points in the request, conversational instance, and application life cycle. Some examples of these capabilities can include:
  - Pre-request execution
  - Post-request execution
  - Conversation start
  - Conversation end
  - Stateless instance reset

- Application deploy
- Application undeploy
- Application reset

- Providing the ability to store and retrieve data from thread specific local context storage associated with the current request. This can allow container-specific information to be associated with the calling context without passing directly on the call stack (since the stack may include user methods where plumbing is best not exposed).
- A uniform mechanism for discovering and initializing a component-pluggable context object.
- A uniform set of internal context APIs for life-cycle event notification.
- Facilities for application life-cycle hooks.

[0035] In some embodiments of the present invention, the control component in a common runtime container framework can support the concept of controls to provide a simplified and common interaction model for how a programmer interacts with external subsystems. This control component can include capabilities such as the following:

- Instantiation, initialization, and hookup of control instances associated with a component. This process can be immediate (at component initialization) or lazy (demand driven) and is selectable on a per-control-instance basis.
- Routing and dispatching of synchronous and asynchronous calls from the component onto the control and of synchronous or asynchronous events from the control onto component event handlers.
- Routing of external system events to internal events trigger by, for example, a control instance. These can be publicly exposed events or internal control events that do not appear as part of the public contract for a control.
- Allowing controls to participate in lifecycle events of the associated component. Examples are conversation start, conversation finish, stateless instance reset, etc.
- Allowing control instances to participate in the overall state management of the containing component.
- Providing factories to support the dynamic instantiation of controls.
- Providing the capability to define internal control (private) callbacks that are not exposed up as events to, for example, the web services developer. In some cases,

these capabilities are used to dispatch internal control events from an external subsystem.

- Participating in annotation validation and metadata construction processes. Controls can throw compile or load time errors, and to build efficient internal representations of attributes for use in dispatch.
- Controlling annotations to indicate if publicly defined events are required (compile or load time error if no event handler is defined), recommended (warning if not present), or optional.

[0036] One embodiment may be implemented using a conventional general purpose or a specialized digital computer or microprocessor(s) programmed according to the teachings of the present disclosure, as will be apparent to those skilled in the computer art. Appropriate software coding can readily be prepared by skilled programmers based on the teachings of the present disclosure, as will be apparent to those skilled in the software art. The invention may also be implemented by the preparation of integrated circuits or by interconnecting an appropriate network of conventional component circuits, as will be readily apparent to those skilled in the art.

[0037] One embodiment includes a computer program product which is a storage medium (media) having instructions stored thereon/in which can be used to program a computer to perform any of the features presented herein. The storage medium can include, but is not limited to, any type of disk including floppy disks, optical discs, DVD, CD-ROMs, micro drive, and magneto-optical disks, ROMs, RAMs, EPROMs, EEPROMs, DRAMs, VRAMs, flash memory devices, magnetic or optical cards, nanosystems (including molecular memory ICs), or any type of media or device suitable for storing instructions and/or data.

[0038] Stored on any one of the computer readable medium (media), the present invention includes software for controlling both the hardware of the general purpose/specialized computer or microprocessor, and for enabling the computer or microprocessor to interact with a human user or other mechanism utilizing the results of the present invention. Such software may include, but is not limited to, device drivers, operating systems, execution environments/containers, and applications.

[0039] The foregoing description of the preferred embodiments of the present invention has been provided for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise forms disclosed. Many modifications and variations will be apparent to the practitioner skilled in the art. Particularly, while the concept "component" is used in the embodiments of the systems and methods described above, it will be evident that such

concept can be interchangeably used with equivalent concepts such as, service, event, control, class, object, bean, and other suitable concepts. Embodiments were chosen and described in order to best describe the principles of the invention and its practical application, thereby enabling others skilled in the art to understand the invention, the various embodiments and with various modifications that are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the following claims and their equivalents.